Core Placement Optimization for Multi-chip Many-core Neural Network Systems with Reinforcement Learning

NAN WU and LEI DENG, University of California, Santa Barbara, USA
GUOQI LI, Tsinghua University, China
YUAN XIE, University of California, Santa Barbara, USA

Multi-chip many-core neural network systems are capable of providing high parallelism benefited from decentralized execution, and they can be scaled to very large systems with reasonable fabrication costs. As multi-chip many-core systems scale up, communication latency related effects will take a more important portion in the system performance. While previous work mainly focuses on the core placement within a single chip, there are two principal issues still unresolved: the communication-related problems caused by the non-uniform, hierarchical on/off-chip communication capability in multi-chip systems, and the scalability of these heuristic-based approaches in a factorially growing search space. To this end, we propose a reinforcement-learning-based method to automatically optimize core placement through deep deterministic policy gradient, taking into account information of the environment by performing a series of trials (i.e., placements) and using convolutional neural networks to extract spatial features of different placements. Experimental results indicate that compared with a naive sequential placement, the proposed method achieves 1.99× increase in throughput and 50.5% reduction in latency; compared with the simulated annealing, an effective technique to approximate the global optima in an extremely large search space, our method improves the throughput by 1.22× and reduces the latency by 18.6%. We further demonstrate that our proposed method is capable to find optimal placements taking advantages of different communication properties caused by different system configurations, and work in a topology-agnostic manner.

CCS Concepts: • **Computer systems organization → Architectures**; **Parallel architectures**; • **Computing methodologies → Reinforcement learning**;

Additional Key Words and Phrases: Multi-chip many-core architecture, neural network accelerator, core placement optimization, machine learning for system

ACM Reference format:
Nan Wu, Lei Deng, Guoqi Li, and Yuan Xie. 2020. Core Placement Optimization for Multi-chip Many-core Neural Network Systems with Reinforcement Learning. *ACM Trans. Des. Autom. Electron. Syst.* 26, 2, Article 11 (October 2020), 27 pages.
https://doi.org/10.1145/3418498

This work was supported in part by NSF Grants No. 1725447, No. 1719160, and No. 1730309.
Authors' addresses: N. Wu, L. Deng (corresponding author), and Y. Xie, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 93106, USA; emails: {nanwu, leideng, yuanxie}@ucsb.edu; G. Li, Department of Precision Instrument, Center for Brain Inspired Computing Research, Tsinghua University, Beijing, 100084, China; email: liguoqi@mail.tsinghua.edu.cn.

1084-4309/2020/10-ART11 $15.00
https://doi.org/10.1145/3418498

1 INTRODUCTION

Machine learning (ML) has been doing wonders in many fields over the past decade, including computer vision [29, 41, 74], speech recognition [26, 30], natural language processing [13, 50, 77], robotics [34, 48], playing video games [57, 84], and many other domains [42, 65, 73]. Current ML models, most of which are deep neural networks (DNNs) and their variants, e.g., multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs), already have high demands for memory and computational resource. As people are seeking better artificial intelligence, there is a trend toward larger, more expressive and more complex models. Aiming at these ever-evolving ML workloads, there arise specialized architectures and accelerators, ranging from those specifically optimized for CNNs (e.g., ShiDianNao [19], Eyeriss [12], and SCNN [61]) to those designed for general-purpose DNN acceleration (e.g., DaDianNao [11], Cambricon-x [95], EIE [27], TPU [39], and DNPU [71]). However, existing DNN systems often diversify in performance, accuracy, and power targets, and it is prohibitively costly to build a dedicated accelerator/architecture for each target. With these considerations, multi-chip many-core neural network systems, which assemble a number of cores into one chip and further interconnect these chips, are attracting increasing attention. These multi-chip many-core systems, from conventional technology such as SpiNNaker [60], TrueNorth [2], Loihi [14], Tianjic [17, 62], and Simba [67], to emerging technology, such as PUMA [3] with memristors, provide high parallelism benefited from decentralized execution, and can be scaled to very large systems with reasonable fabrication costs.

Usually there are two major steps to map an application or a neural network (NN) model to a many-core system. In the first step, the computational graph is partitioned into small groups that are compatible to the computation capability of each core [20, 62, 67], in which we refer these small groups as **logic cores**, since some of them are logically connected with demand of communication and they are not yet placed on physical chips (see Figures 1(a) and 1(b)). Then in the second step, these logic cores are placed onto **physical cores**—such process is defined as the **core placement** (see Figure 1(c)). As multi-chip many-core systems scale up, communication costs would be a concern in these decentralized systems, and partitioning and placement of computation onto cores heavily impact the efficiency of on-chip and off-chip communication [2, 46, 67, 82]. Some work has been proposed to optimize the partitioning in the first step aiming to reduce required communication between logic cores. For example, Urgeses et al. [82] present a partitioning methodology to optimize network traffic for spiking neural networks on neuromorphic many-core platforms; HyPar [75] searches a partition that minimizes the total communication of DNNs on an accelerator array. When it comes to the second step, there is a series of heuristic-based investigations in mapping applications, especially multi-media workloads, to 2D-mesh NoC architectures [32, 33, 47, 58, 64, 68, 69].

However, there are two principal issues still unresolved in the core placement step. First, these previous approaches all target on general-purpose many-core systems in *a single chip*, whereas in decentralized multi-chip many-core systems, the communication related problem is caused by not only the demand for communication among different cores but also *the non-uniform, hierarchical on/off-chip communication capability*. Second, the *scalability* is a concern of these heuristic-based methods, where they mainly handle systems with tens of cores and have limited design space exploration capabilities. It has been noticed that the computation complexity of these heuristic-based methods grows drastically [32] as systems scale up. To find an optimal core placement is an *NP-hard* [24] problem and the search space of this problem *grows factorially* with the system size.

To this end, we propose a reinforcement-learning (RL)-based method learning to optimize core placement with neural networks. This method, as shown in Figure 8, takes into account information of the environment by performing a series of trials (i.e., placements) to understand which

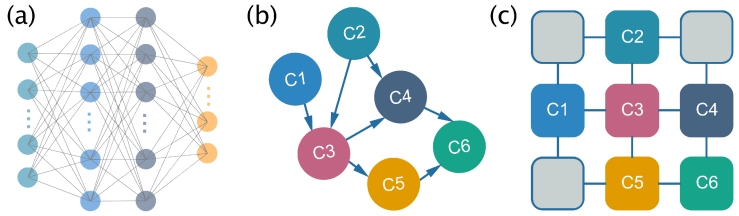

Fig. 1. NN mapping: (a) the original NN; (b) the NN partitioned into logic cores; (c) logic cores placed onto physical cores.

logic core should be placed on which physical core so that the overall latency can be optimized. The specific algorithm leveraged in our work is the deep deterministic policy gradient (DDPG) [72], since the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient, leading to a faster training process. To guarantee sufficient exploration in the search space, we employ the off-policy deterministic actor-critic, a variant of DDPG. Key to our method is the use of CNNs to extract spatial features of different placements, and the latency of the predicted placement is then used as the reward signal to optimize parameters in the networks. Evaluations indicate that compared with a naive sequential placement, the proposed RL-based method achieves 1.99× increase in throughput and 50.5% reduction in latency; compared with the simulated annealing [83], an effective technique to approximate the global optima in an extremely large search space, our method improves the throughput by 1.22× and reduces the latency by 18.6%.

The specific contributions of our work are as follows:

- We consider DNN inference in multi-chip many-core systems, where a hierarchical pipeline is implemented, i.e., a block-by-block streaming pipeline for intra-frame dataflow and a stage-based pipeline for inter-frame dataflow. Then, we formulate the core placement optimization problem.
- We propose an RL-based method to automatically optimize core placement through DDPG, taking into account information of the environment by performing series of trials and using CNNs to extract spatial features of different placements.
- We evaluate our proposed method on multiple workloads: AlexNet [41], VGG16 [74] and ResNet50 [29]. On the geometric average, it achieves 50.5%, 38.4%, and 18.6% reduction in the overall latency and improves the throughput by 1.99×, 1.61×, and 1.22×, compared with the sequential placement, the random search and the simulated annealing, respectively.
- We demonstrate that our proposed method is capable to find optimal placements taking advantages of different communication properties under different system configurations; it can also work in a topology-agnostic manner, which is showcased by simple examples in several other topologies, such as 2D torus, HNoC [10], and dragonfly [40].

2 BACKGROUND

2.1 DNN Workloads

There are multiple variants of DNNs, including MLPs, CNNs, RNNs, and so on. As illustrated in Figure 2, a convolutional (CONV) layer can be considered as a seven-dimensional nested loop on input activations (IA), weights (W), and output activations (OA), with the batch size B, the height H and the width T of OA, the number of output channels K, the number of input channels C, the

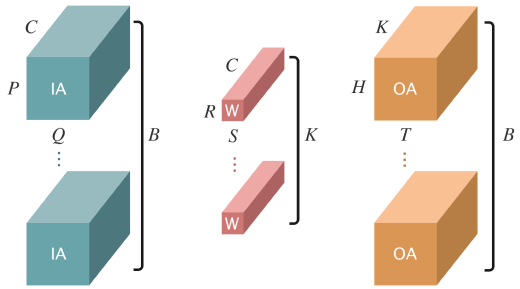

Fig. 2. The seven-dimensional nested loop of convolutional layers, on input activations (IA), weights (W), and output activations (OA).

height R and the width S of the weight kernel. Similar formulations can also be applied to fully connected (FC) layers, which are widely used in MLPs and are essential components in DNNs.

2.2 Multi-chip Many-core Architectures

Multi-chip many-core architectures, which are broadly employed to build up neuromorphic systems, arise with the era of cognitive computing that demands systems capable of processing massive amounts of multi-sensory data. Among the issues to be solved with top priority in these systems, real-time operation, low-power consumption and scalability are those attracting the most attention, and thus parallel architectures working in a decentralized way are developed. There are several notable examples. SpiNNaker [60], which can model up to one billion neurons and one trillion synapses, integrates 18 ARM cores per chip and is able to scale to a system with 65,536 chips. The TrueNorth chip [2] from IBM organizes 4096 neurosynaptic cores by 2D mesh, containing one million digital neurons and 256 million synapses; multiple TrueNorth chips can be further interconnected to build more complex TrueNorth systems. Loihi [14] from Intel also utilizes the 2D mesh topology to comprise 128 neuromorphic cores and three embedded x86 processor cores on a single chip, and off-chip communication interfaces are used to connect other chips.

As the variety of DNN workloads increases and the performance, energy and power targets diversify in different workloads, the concerns previously discussed in neuromorphic systems also appear in the design of deep-learning accelerators, and it is prohibitively costly to design a dedicated accelerator for each target of each workload. One potential resolution is to employ the multi-chip-module-based (MCM-based) integration, just as Simba [67], which is a scalable deep-learning inference accelerator with MCM-based architectures, providing a promising approach for building large-scale systems. In Simba, it is noticed that the disparity in latency and bandwidth between on-chip and on-package (off-chip) communication leads to significant latency variability across chiplets, based on which Simba optimizes workload partitioning and data placement to mitigate the inter-chiplet communication overheads, through a random search algorithm to select good mappings and placements.

In terms of the integration of both neuromorphic primitives (e.g., spiking neural networks) and DNNs, there is the Tianjic chip [17, 62], a multi-chip many-core architecture providing a hybrid platform toward artificial general intelligence. The Tianjic chip, consisting of 156 functional cores, shows significant improvement in both throughput (1.6× to 10^2×) and power efficiency (12× to 10^4×) compared with the GPU. Promising potentials are demonstrated that an unmanned bicycle can achieve autonomous riding by equipped with a single Tianjic chip running multiple different NN models.

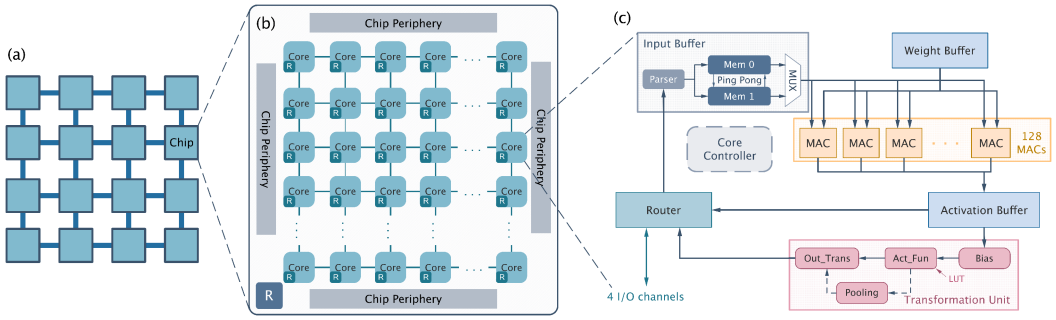

Fig. 3. Illustration of a typical multi-chip many-core neural network architecture: (a) the multi-chip system, (b) the many-core chip, and (c) one single core.

In Figure 3, we take Tianjic as an example to illustrate the typical multi-chip many-core architecture. Usually, multiple chips (e.g., 4×4 in Figure 3(a)) can be interconnected through off-chip links such as low-voltage differential signaling (LVDS) [9], SerDes [7] and ground-referenced signaling (GRS) [89, 96]. As illustrated in Figure 3(b), each chip includes an array of functional cores arranged by a 2D mesh network-on-chip (NoC), an on-chip router for off-chip communication and essential chip peripherals. Figure 3(c) details the micro-architecture of each core, which leverages parallel multiplier-and-accumulator (MAC) units for efficient and flexible computation and contains peripheral processing circuits, such as an input buffer, a weight buffer, an activation buffer, a transformation unit, a core controller and a router. The input buffer provides input activations for MACs, where the ping-pong buffer scheme is used to decouple writes by the router and reads by MACs. The MACs conduct most of the computation, multiplying the input activations read from the input buffer with the weights stored in the distributed weight buffer to implement vector-matrix multiplications (VMMs). The activation buffer is used to buffer either intermediate activations or results that do not need to go through the transformation unit. The transformation unit is responsible for adding bias, non-linear activation functions, possible pooling operations and generating output activations, and it finally sends the results to the router. The core controller manages the overall timing sequence, and whether to enable these MACs or the transformation unit.

The multi-chip many-core architecture is essentially a spatial architecture, since there is no off-chip DRAM and all weights must be stored on chip, which is different from common deep-learning accelerators. As such, it often uses a weight-stationary dataflow: Weights remain in the weight buffer of each core and are reused across iterations, while new input activations are injected at each time phase.

2.3 Reinforcement Learning

In the standard setting of RL [78], an agent interacts with an environment $\mathcal{E}$ over a number of discrete time steps, as shown in Figure 4. At each time step t, the agent gets a state s_t from the *state space* $\mathcal{S}$, and selects an action a_t from the *action space* $\mathcal{A}$ according to its policy π that is a mapping from the state s_t to the action a_t. In return, the agent receives the next state s_{t+1} and a scalar reward $r_t : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. This process continues until the agent reaches a terminal state after which the process restarts.

The accumulated rewards after the time step t can be expressed as

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{1}$$

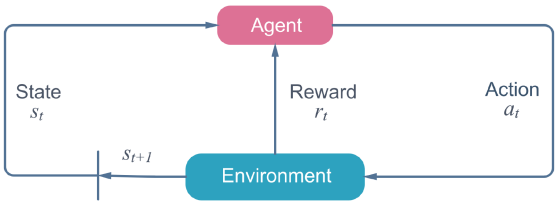

Fig. 4. A typical framing of RL.

where $\gamma \in (0, 1]$ and γ is the *discount factor*. The state-action value $Q_\pi(s, a)$ is represented by

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a], \tag{2}$$

which is the expected return after selecting action a at state s with policy π. Similarly, the state value $V_\pi(s)$ is defined as

$$V_\pi(s) = \mathbb{E}_\pi[R_t | s_t = s], \tag{3}$$

which is the expected return starting from state s by following policy π. The goal of the agent is to maximize the expected return for every state s.

There are two general types of methods for RL: value-based methods and policy-based methods. In value-based methods, the state-action value function $Q_\pi(s, a)$ is approximated by either tabular approaches or function approximations, and at each state s the agent always selects the optimal action a^* that can bring the maximal state-action value $Q_\pi(s, a^*)$. One well-known example of value-based methods is Q-learning [86]. As for policy-based methods, they directly parameterize the policy $\pi_\theta(s, a)$ and update the parameters θ by performing gradient ascent on $\mathbb{E}_\pi[R_t]$. One example is the REINFORCE algorithm [88]. In standard REINFORCE, the policy parameters θ are updated in the direction of $\nabla_\theta \log \pi_\theta(s_t, a_t) Q_\pi(s_t, a_t)$, which is an unbiased estimate of $\nabla_\theta \mathbb{E}_\pi[R_t]$.

RL is modeled based on Markov decision process [6], and thus it is talented to handle sequential decision-making processes. By embedding optimization goals into reward functions, RL agents are able to figure out optimal solutions.

3 APPROACH

3.1 Formulation of Core Placement Optimization

For simplicity and clarity, we consider the spatial mapping with a weight-stationary dataflow for DNN inference.

3.1.1 Mapping Neural Networks to Logic Cores. Taking advantages of model parallelism, there have been several state-of-the-art DNN tiling techniques [11, 39, 61, 91] proposed to partition weights in the spatial mapping, based on which we partition DNN weights uniformly along the input channel C and the output channel K. Figure 5 illustrates the uniform partitioning of CONV and FC layers. In the decentralized many-core system, outputs of VMM cores will be delivered to other cores for cross-core partial-sum reduction, referred to as vector-vector-accumulation (VVA), if handling with large neural networks. We decouple the execution of VMM and VVA to different cores to ease the timing implementation.

We further optimize the uniform partitioning by two steps: first, we balance the computation required on each core, to avoid over-busy or idle cores; second, we consider the trade-off between the exploitation of computation parallelism and the communication/synchronization costs. Figure 6 shows the breakdown of logic cores for different models. Since CONV layers are often bound by computation while FC layers are often bound by memory, more logic cores are assigned for CONV layers to balance the computation.

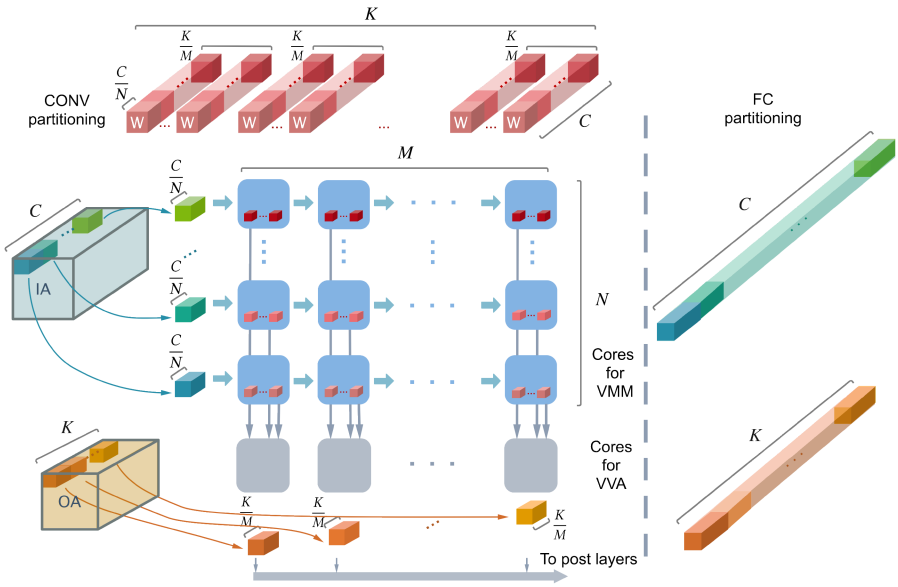

Fig. 5. The uniform partitioning of CONV or FC layer, where the red tensors on the top represent weights (W), the green tensors in the middle represent input activations (IA), and the orange tensors at the bottom represent the output activations (OA).

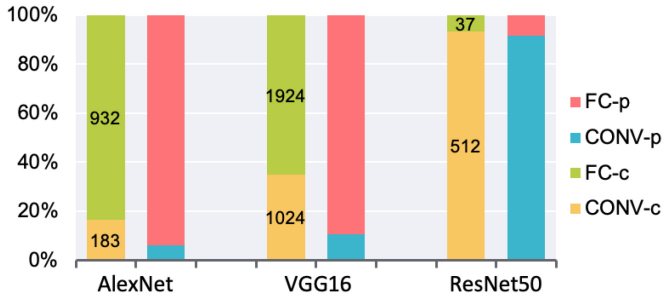

Fig. 6. The breakdown of parameters (denoted with -p) and logic cores (denoted with -c) for CONV and FC layers in different models, with the number of logic cores marked for each model.

3.1.2 Core Placement. Consider a set of logically connected cores consisting of Z logic cores $\{Core_1, Core_2, \ldots, Core_Z\}$, and a set of D available physical cores $\{V_1, V_2, \ldots, V_D\}$ connected in a specific topology (where $Z \leq D$). A placement $\mathcal{P} = \{p_1, p_2, \ldots, p_Z\}$ is an assignment of a logic core $Core_i$ to a physical core p_i, where $p_i \in \{V_1, V_2, \ldots, V_D\}$ and $\forall i \neq j$, there is $p_i \neq p_j$.

In each single frame, it is possible to implement a streaming pipeline across multiple CONV layers to take advantage of inter-layer parallelism, because each convolution operation only needs part of input activations, whereas for FC layers one output activation cannot be generated until all input activations are ready, indicating that there only exists intra-layer parallelism. Based on this execution difference, we consider a hierarchical pipeline execution. For the inter-frame execution, a stage-based pipeline is used to decouple the computation of CONV and FC layers, leveraging better parallelism. Accordingly, we place the logic cores for CONV and FC layers in different regions of the multi-chip many-core system, and *optimize their core placement processes separately* by masking unused regions.

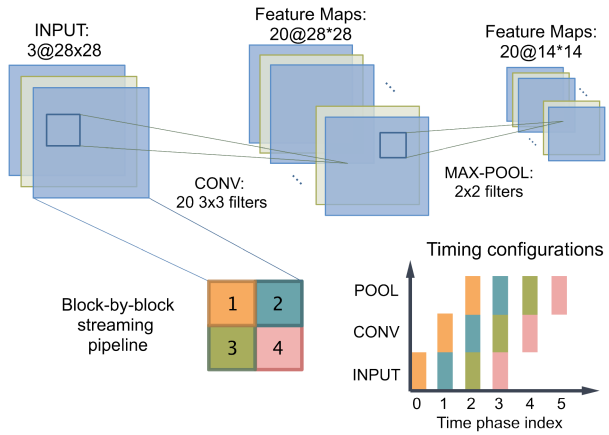

Fig. 7. An example of the block-by-block streaming pipeline execution and its corresponding timing configurations.

Then in the intra-frame execution, instead of the row-by-row streaming pipeline [16] in which logic cores have more and more idle time as layer propagates, we employ the block-by-block streaming pipeline for CONV layers to make better utilization of resources. As depicted in Figure 7, we showcase a block-by-block streaming pipeline and its corresponding timing configurations by an example of the MNIST dataset [43], where each input frame is divided into four blocks. In this example, there is a three-stage streaming pipeline, with one time phase for one pipeline stage; to guarantee system functionality, the time phase, which contains both the computation latency as well as the communication latency, should be long enough to cover the pipeline stage that consumes the maximum latency. Similar analogy is used in FC layers, where the pipelined execution is applied layer by layer.

Assume there is an $\mathcal{F}$-stage streaming pipeline for intra-frame execution, we use $T(k|\mathcal{P})$ to denote the latency of the kth pipeline stage for a given placement $\mathcal{P}$. By minimizing the maximum latency among all stages, the overall latency and throughput can be optimized. Therefore, the optimization goal is

$$\mathcal{P}^* = \underset{\mathcal{P}}{\operatorname{argmin}} \{L(\mathcal{P})\}, \tag{4}$$

where $L(\mathcal{P}) = \max_k \{T(k|\mathcal{P})\}$ for $k = 1, 2, \ldots, \mathcal{F}$.

3.2 Learning Core Placement with Deep Deterministic Policy Gradient

Overview. Figure 8 presents the overview of the RL-based core placement optimization. The agent attempts to learn an optimal core placement to minimize the overall latency, and the environment gives feedback to the agent by different rewards to encourage or punish the agent according to its behaviors. Through interactions with the environment, the agent is able to learn and figure out the optimal policy.

We build the core placement problem as a Markov decision process. At the beginning of each trial, no assignment has been generated and all physical cores are available. At each time step t, with the observation of currently available physical cores and unplaced logic cores, which is referred as the *state* s_t in the state space $\mathcal{S}$, a placement of a couple of logic cores will be generated, which is referred as the *action* a_t in the action space $\mathcal{A}$. With this action a_t, corresponding physical cores are occupied by these assigned logic cores, and the state s_t is updated to the state s_{t+1}. The placement of logic cores is generated sequentially according to the index and the reward

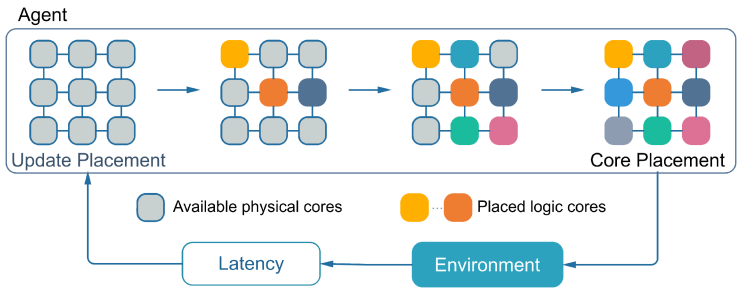

Fig. 8. Overview of the RL-based core placement optimization.

is provided at each time step. It is notable that from our simulation results, different placement orders have little influence on learning performance, as the agent can adjust its behaviors during interactions with the environment, mitigating the effects caused by different orders. When all logic cores have been placed onto physical cores, the overall performance of this placement is measured by the maximum latency among all pipeline stages, i.e., $L(\mathcal{P})$, which is then used to derive the final reward of this placement. These rewards, together with information from states, actions, and state-action values, are combined to train the agent and update following placements.

Representations of Core Placement Optimization. The mathematical representations of the state, the action and the reward of core placement optimization are detailed in this part as follows.

- *Representation of Core Placements (i.e., the states).* Among most multi-chip many-core architectures for neural networks, the 2D mesh topology is the mainstream for both intra-chip and inter-chip interconnect. Simultaneously, the connectivity information that we mainly consider is the communication characteristic, and so we prefer the matrix representation of the placement, simpler and more intuitively appropriate. In our formulation, the state s_t is represented by a 2D matrix to encode the current placement status, which includes the information required by the agent to make decisions, as shown in the upper part of Figure 9. In this illustration, a 3×3 chip array with 2×2 cores per chip is represented by a 6×6 matrix, where the available physical cores are denoted by zero and occupied physical cores are denoted by the indexes of their assigned logic cores. In general, the state of the current placement on a multi-chip many-core system composed of a $row_{chip} \times col_{chip}$ chip array with $row_{core} \times col_{core}$ cores per chip can be denoted as a $(row_{chip} \times row_{core})$-by-$(col_{chip} \times col_{core})$ matrix.
- *Representation of Assigning Placements (i.e., the actions).* Given that the current core placement is uncompleted, the action is defined as assigning a placement of z unplaced logic cores, which is encoded as $[x_1, y_1, x_2, y_2, \ldots, x_z, y_z]$, with (x_i, y_i) representing the physical coordinate on which a logic core will be placed.
- *Representation of the Reward Function.* We empirically find that defining the reward at the time step with a completed placement as $r_t = \sqrt{\mathcal{B}} - \sqrt{L(\mathcal{P})}$, where $\mathcal{B}$ is the latency of the best placement found by the random search, makes the learning process more robust. With this definition, placements that lead to better latency are encouraged by positive rewards, the shorter latency the more reward received; while placements with worse latency are penalized by negative rewards. For those time steps at which one placement is not completed, the reward is defined as $r_t = 0$.

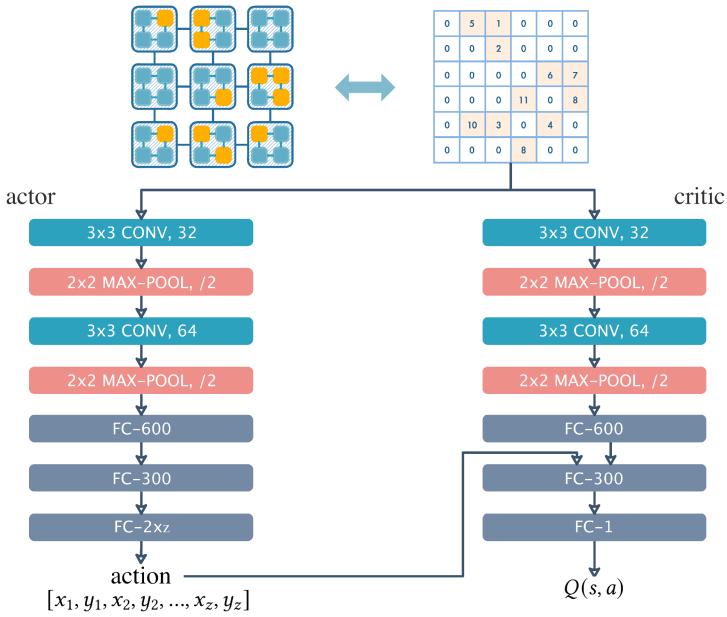

Fig. 9. DNN structure of the RL-based agent: the actor, and the critic.

Deterministic Policy Gradient (DPG). Policy gradients have been broadly applied under different RL scenarios, where the basic idea is to directly parameterize the policy via a probability distribution $\pi_\theta(s, a) = \mathbb{P}(a|s; \theta)$ that stochastically takes the action a given the state s according to the parameters θ. If we define the discounted state distribution [79] by

$$\rho_\pi(s') := \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{P}(s_t = s'|s_0 = s, \pi) \mathbb{P}(s_0 = s) \mathrm{d}s, \tag{5}$$

then the expected return can be expressed as

$$\begin{aligned} \mathcal{J}(\pi_\theta) &= \mathbb{E}_{s\sim\rho_\pi, a\sim\pi_\theta} \left[\sum_{k=0}^{\infty} \gamma^k r_k \right] \\ &= \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s, a) Q_\pi(s, a) \mathrm{d}a\mathrm{d}s, \end{aligned} \tag{6}$$

where $Q_\pi(s, a)$ is defined in Equation (2) and the discount factor $\gamma \in (0, 1]$.

To maximize the expected return of a stochastic policy, the corresponding stochastic policy gradient algorithm should update the parameters θ by performing gradient ascent on the expected return, i.e., adjusting the parameters θ in the direction of $\nabla_\theta \mathcal{J}(\pi_\theta)$, where

$$\begin{aligned} \nabla_\theta \mathcal{J}(\pi_\theta) &= \nabla_\theta \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s, a) Q_\pi(s, a) \mathrm{d}a\mathrm{d}s \\ &= \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} Q_\pi(s, a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho_\pi, a\sim\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s, a) Q_\pi(s, a) \right]. \end{aligned} \tag{7}$$

Whereas in our work, instead of the stochastic policy, we give attention to the deterministic policy [72]. We propose to train a deterministic policy $\mu_\theta(s) : \mathcal{S} \rightarrow \mathcal{A}$, which is a deterministic

mapping from the current placement status s_t to the action a_t – the placement assignment of unplaced logic cores. With the deterministic policy, the core placement process can be optimized by maximizing

$$\begin{aligned} \mathcal{J}(\mu_\theta) &= \mathbb{E}_{s\sim\rho_\mu}\left[\sum_{k=0}^{\infty}\gamma^k r_k\right] \\ &= \int_{\mathcal{S}} \rho_\mu(s) Q_\mu(s,\mu_\theta)\mathrm{d}s. \end{aligned} \tag{8}$$

Then the deterministic policy gradient is derived as

$$\begin{aligned} \nabla_\theta \mathcal{J}(\mu_\theta) &= \nabla_\theta \int_{\mathcal{S}} \rho_\mu(s) Q_\mu(s,\mu_\theta)\mathrm{d}s \\ &= \int_{\mathcal{S}} \rho_\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q_\mu(s,a)|_{a=\mu_\theta(s)} \\ &= \mathbb{E}_{s\sim\rho_\mu}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q_\mu(s,a)|_{a=\mu_\theta(s)}\right]. \end{aligned} \tag{9}$$

From the practical perspective, the cardinal reason of applying deterministic policy gradient rather than stochastic policy gradient is that the stochastic policy gradient should be estimated by the integration over both the state space and the action space, as shown in Equation (7); while the deterministic policy gradient only needs to integrate the state space as in Equation (9), indicating that it can be estimated more efficiently and leads to a faster learning process, especially for a large action space, which is our case.

Aiming at combining the benefits of both policy-based methods and value-based methods, we employ the off-policy deterministic actor-critic (OPDAC) [72], a variant of DPG, which consists of two components: the *critic* and the *actor*. The critic estimates the action-value function $Q_w(s,a) \approx Q_\mu(s,a)$ by adjusting parameters w based on Q-learning, and the actor learns the deterministic policy $\mu_\theta(s)$ by ascending the gradient of the action-value function.

To improve the sample efficiency of the learning process, we apply the experience replay taking advantages of past experiences, which is implemented by a replay buffer storing tuples (s_t, a_t, r_t, s_{t+1}) from history trajectories. To sufficiently explore the large search space, we add Ornstein-Uhlenbeck noise [81] to the action space, which is multiplied by a fading factor as the training process proceeds.

DNN Structure of the RL-based Agent. The structure of the RL-based agent is depicted in Figure 9, where both the actor and the critic have similar network structures. The input to these DNNs is the current state of the placement being predicted, which includes physical cores either currently available or already assigned with logic cores. Then the actor outputs the action in vector, and the critic generates the state-action value in scalar. Since the state-action value is a function of both the current state and the action being taken, the output of the actor is merged to the critic after its first FC layer. We employ CONV layers followed with max-pooling layers to extract spatial features of various placements, because there are some similarities between the core placement analysis and image analysis, on which CONV layers usually perform well. The local response normalization is applied after each pooling layer, and the batch normalization is applied after each FC layer. The activation function is ReLU for all layers, except for the output of the actor, which uses *tanh* to bound actions to the size of multi-chip many-core systems. Since the outputs of the actor network are in continuous values, we apply the *floor* function to derive placement locations, i.e., finding the closest integers that are no larger than the outputs. If there is a contradiction between the currently being placed core and an already placed core, then the current core will

ALGORITHM 1: Deep deterministic policy gradient for core placement optimization

Initialize parameters θ for the actor and w for the critic;
Initialize the episode counter $i \leftarrow 0$;
Initialize the best core placement $\mathcal{P}_{best} \leftarrow \mathcal{P}_{baseline}$;
while $i < episode_{max}$ **do**
  $t \leftarrow 0$; // The time step counter.
  Initialize state $s_t \leftarrow$ an empty placement;
  while $t < step_{max}$ **do**
    Perform action a_t based on policy $\mu_\theta(s_t)$;
    Get updated placement s_{t+1};
    if *all logic cores have been placed* **then**
      Receive the reward $r_t = \sqrt{\mathcal{B}} - \sqrt{L(s_{t+1})}$;
      Add (s_t, a_t, r_t, s_{t+1}) into replay buffer;
      if $L(s_{t+1}) < L(\mathcal{P}_{best})$ **then**
        $\mathcal{P}_{best} \leftarrow s_{t+1}$;
      end
      Clear state $s_{t+1} \leftarrow$ an empty placement;
    else
      Receive the reward $r_t = 0$;
      Add (s_t, a_t, r_t, s_{t+1}) into replay buffer;
    end
    Update θ and w according to Equations (10)–(12);
    $t \leftarrow t + 1$;
  end
  $i \leftarrow i + 1$;
end
return $\mathcal{P}_{best}$;

be placed to the position that has the minimum Manhattan distance to its originally intentional position. If there are multiple available candidates, then we choose the first one found.

The network parameters are learned by Adam optimizer based on the estimation of Equation (9), which is computed by sampling a minibatch of size $\mathcal{K}_{mb}$ from the replay buffer, leading to the updates of parameters as follows:

$$\delta_i = r_i + \gamma Q_w(s_{i+1}, \mu_\theta(s_{i+1})) - Q_w(s_i, a_i), \tag{10}$$

$$w_{t+1} = w_t + \alpha_w \cdot \frac{1}{\mathcal{K}_{mb}} \sum_{i=t_1}^{t_{\mathcal{K}_{mb}}} \delta_i \nabla_w Q_w(s_i, a_i), \tag{11}$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \cdot \frac{1}{\mathcal{K}_{mb}} \sum_{i=t_1}^{t_{\mathcal{K}_{mb}}} \nabla_\theta \mu_\theta(s_i) \nabla_a Q_w(s_i, a_i)|_{a=\mu_\theta(s)}, \tag{12}$$

where α_w and α_θ are learning rates of the critic and the actor, respectively, and $i \in \{t_1, \ldots, t_{\mathcal{K}_{mb}}\}$.

The entire procedure of core placement optimization with deep deterministic policy gradient is summarized in Algorithm 1.

4 EXPERIMENT

In this section, we present the experiment setup, the baseline, and the analysis of the results.

Table 1. Simulation Configuration Parameters

System	Number of Chips	4×4
	Off-chip Interconnect	GRS
	Off-chip Interconnect Bandwidth	100 GB/s/chip
Chip	Number of Cores	16×16
	Technology	UMC 28-nm HLP
	NoC Interconnect Bandwidth	64 GB/s/core
	Core Frequency	400 MHz
Core	Weight Buffer Size	64 KB
	Input+Activation Buffer Size	64 KB
	Number of MACs	128
	MAC Width	8b
	Input/Weight Precision	8b
	Partial-sum Precision	32b

4.1 Experiment Setup

We build an in-house simulator for the typical multi-chip many-core architecture illustrated in Figure 3. The overall system consists of a 4×4 chip array with 16×16 cores per chip, with the off-chip interconnect assumed as GRS [89, 96]. Generally, the routing is based on the minimal path, with X-Y routing for both NoC and off-chip communication. Although all chips are functional in the multi-chip many-core system, different workloads may occupy different number of chips/cores, since in these spatially weight-stationary mappings, the number of cores consumed is kind of proportional to the model size. Configuration parameters are summarized in Table 1, which are collected from existing multi-chip many-core architectures [17, 62, 67, 89, 96]. As for hyperparameters in our RL-based approach, the learning rates of the actor and the critic are set as $\alpha_\theta = 0.0002$ and $\alpha_w = 0.001$, with the discount factor $\gamma = 0.98$. In each epoch the actor predicts 30 placements and the size of minibatch is $\mathcal{K} = 64$.

We consider DNN workloads of AlexNet [41], VGG16 [74], and ResNet50 [29], which are representative deep-learning models, and evaluate the latency when the batch size is one and the throughput when the batch size is much larger. The overall latency is derived according to the latency of each time phase, which is measured by adding the computation latency (i.e., the cycles required for computation) and the communication latency. As mentioned in Section 3.1.2, we place the logic cores for CONV and FC layers in different regions of the multi-chip many-core system, and *optimize their core placement processes separately*.

4.2 Baselines

Our RL-based approach (denoted by DDPG) is evaluated with the following placement methods.

- *Sequential placement* (denoted by BS): logic cores are placed sequentially along with the indexes of physical cores (first chip index, then core index).
- *Random search* (denoted by RS): one million placements are sampled randomly, and the best placement found during the random search is selected.
- *Simulated annealing* [83] (denoted by SA): SA is an effective technique to approximate the global optima in an extremely large search space, with the procedure detailed in Algorithm 2. The cooldown factor is set as 0.99; the initial temperature T_0 and the ending temperature T_{end} are chosen according to the application such that around one million

ALGORITHM 2: Simulated annealing for core placement optimization

Randomly generate initial core placement $\mathcal{P}_{current} \leftarrow \mathcal{P}_0$;
Initialize the best core placement $\mathcal{P}_{best} \leftarrow \mathcal{P}_0$;
Initialize temperature $T \leftarrow T_0$;
while $T > T_{end}$ **do**
  $iter \leftarrow 0$;
  while $iter < iteration_{max}$ **do**
    Select a placement $\mathcal{P}_{new} \in \mathcal{N}(\mathcal{P}_{current})$;
    /* A neighbor placement to current placement. */
    if $L(\mathcal{P}_{new}) < L(\mathcal{P}_{current})$ **then**
      $\mathcal{P}_{current} \leftarrow \mathcal{P}_{new}$;
      if $L(\mathcal{P}_{new}) < L(\mathcal{P}_{best})$ **then**
        $\mathcal{P}_{best} \leftarrow \mathcal{P}_{new}$;
      end
    else
      $\Delta = L(\mathcal{P}_{new}) - L(\mathcal{P}_{current})$;
      Accept $\mathcal{P}_{current} \leftarrow \mathcal{P}_{new}$ with probability $\mathbb{P} = e^{-\Delta/T}$;
    end
    $iter \leftarrow iter + 1$;
  end
  $T \leftarrow \alpha \times T$;
  /* $0 < \alpha < 1$, the cooldown factor. */
end
return $\mathcal{P}_{best}$;

placements would be searched; and the neighborhood function $\mathcal{N}(\mathcal{P}_{current})$ indicates that the placement of 1% of logic cores in $\mathcal{P}_{current}$ will be randomly changed.

4.3 Analysis of Core Placements Optimized by DDPG

Figure 10 compares both the latency and the throughput of different core placement methods for AlexNet, VGG16 and ResNet50 workloads. DDPG achieves significant improvements among all considered workloads, especially for VGG16 that has the largest model size, where DDPG reduces the overall latency by 67.4%, 51.7%, and 23.2%, and improves the throughput by 3.06×, 2.07×, and 1.30×, compared with BS, RS, and SA, respectively. Generally, there is usually a larger optimization space for large models, because they are often mapped onto more logic cores, resulting in a larger search space, just as aforementioned that the search space of core placement optimization *grows factorially* with the system size. In addition, more conspicuous improvements are shown in FC layers than those in CONV layers, since the inter-layer connections are denser in FC layers; one exception comes from the FC layers in ResNet50, whose small layer size leads to a relatively small search space as well as a small optimization space. Furthermore, the communication demand is usually related to the size of feature maps, the number of input and output channels, and whether there exist bypass connections in the workload neural networks, and so on, indicating that the more complex the network structure is, the higher the communication demand is often required, and thus the more essential it is to conduct the core placement optimization. Stronger improvements are displayed by DDPG in VGG16 and ResNet50, since both of them have more complicated network structures than that of AlexNet. Considering all the workloads, on the geometric average, 50.5%, 38.4%, and 18.6% reductions in the overall latency are achieved by DDPG, with the throughput improvement of 1.99×, 1.61×, and 1.22×, compared with BS, RS, and SA, respectively.

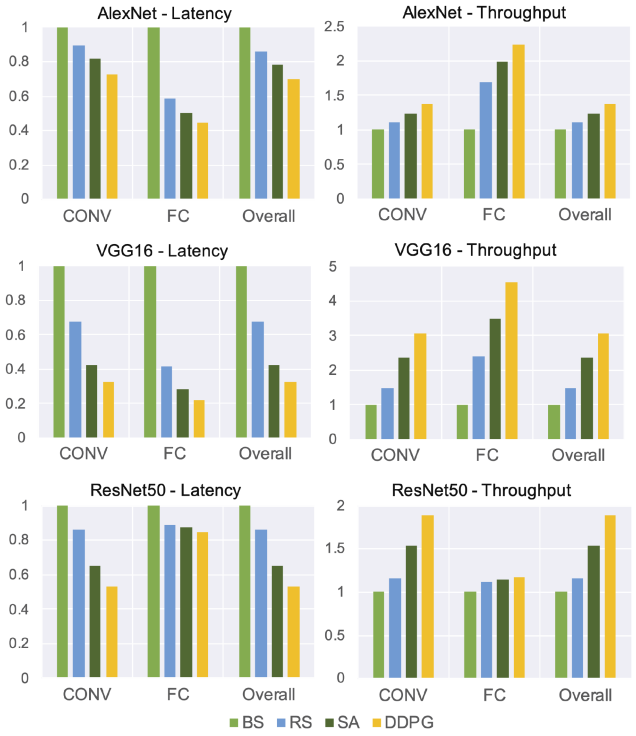

Fig. 10. Latency and throughput of different placement methods, both of which are normalized to BS (the baseline).

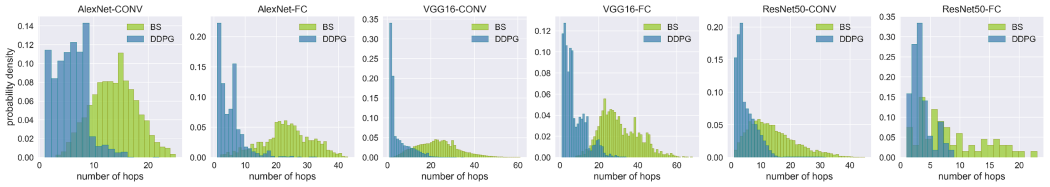

Fig. 11. Hop distributions of BS- and DDPG-based core placement optimization.

Notably, under the scenario of extremely large search spaces, DDPG substantially outperforms SA. In SA, new placements at each time are randomly picked from the neighborhood of the current placement, and whether or not to accept a new placement is dependent on the latency, or to be more specific, the objective function, ignoring past experiences and introducing unreliability to the search process. In contrast, DDPG proactively explores the search space. Learning from different rewards received during the exploration, DDPG extracts useful spatial features from various placements, to avoid defective placements and further encourage trials to approach optimums. Through the leverage of experience replay, past experiences can be consolidated into the training process, thus stabilizing the overall learning and search process.

To show more insights of core placements found by DDPG, Figure 11 depicts the hop distributions before and after DDPG-based core placement optimization, in which the averaged hop distances in the CONV and FC layers of AlexNet, VGG16, and ResNet50 are reduced by 2.5×, 4.5×, 4.6×, 4.3×, 2.9×, and 2.8×, respectively. The geometric average reduction in hop distance is 3.5×.

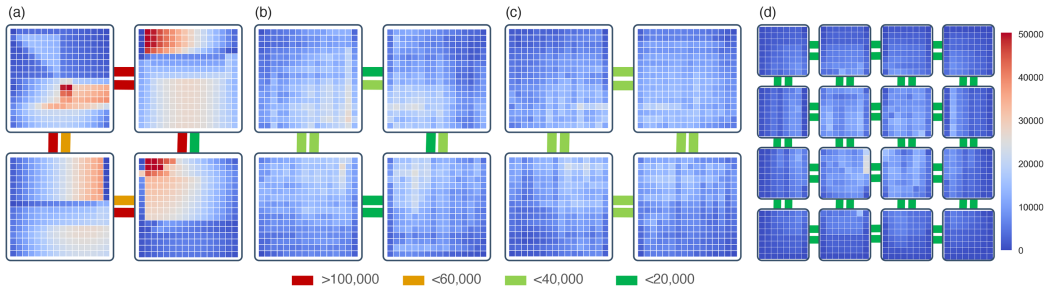

Fig. 12. The distribution of total number of packets transferred through each core per time phase, and the total number of packets delivered by each off-chip link per time phase, for core placements of VGG16-CONV: (a) placed by BS, (b) optimized by DDPG, (c) with doubled off-chip bandwidth optimized by DDPG, and (d) with less cores per chip optimized by DDPG.

This indicates that long data paths are significantly shrunken and cores that are logically connected are tended to be placed on the nearby regions, reducing long travel time of data as well as removing potential congestions. Additionally, with the reduced hop distances, the active communication power consumption can also be implicitly reduced.

Figures 12(a) and 12(b) show the communication traffic of placements optimized by BS and DDPG. For BS, there are multiple extremely busy cores for on-chip communication, and there are several off-chip links having heavy communication workloads; whereas after the optimization by DDPG, both the on-chip and the off-chip communication are balanced: the unnecessary off-chip communication is removed to the on-chip communication that usually consumes lower costs, and the traffic of busy cores is spreaded to those relatively idle cores. DDPG ensures that the off-chip traffic is low enough to avoid congestion delay, thus improving the latency.

As for cost evaluations of DDPG, in terms of the number of placements evaluated, our method can converge at around 300K $\sim$ 400K placements, conspicuously surpassing the best placements found by either RS or SA with one million searched placements; in terms of the running time, since DDPG is a learning-based method, it definitely consumes longer time per placement evaluated, but shorter time in total to find a better placement. Even if the training time is longer than other approaches, this is a one-time cost and can be amortized by every future inference on chip once the placement is completed during compilation.

4.4 Strong Learning Capability of DDPG

Here, we discuss the strong learning capability of our proposed DDPG-based core placement optimization that can make better use of different communication configurations; we also demonstrate that DDPG has great versatility that can work for several other topologies such as 2D torus, HNoC [10] and dragonfly [40], in a topology-agnostic manner.

Making better use of communication configurations. To examine the off-chip communication bandwidth sensitivity of DDPG-based core placement optimization, we adjust the off-chip bandwidth to 1.5× and 2.0× of its original configuration, and apply core placement optimization under each configuration. It is undoubted that given the increased bandwidth there should appear reductions in latency, even if the original placement is not modified according to these changed communication properties. To demonstrate the influence coming from the increased off-chip communication bandwidth, we fix placements that are optimized under the original configuration and only make changes in the off-chip communication bandwidth; as shown in Figure 13, there achieves less than 10% reduction in latency. Then in Figure 14, we compare the optimized

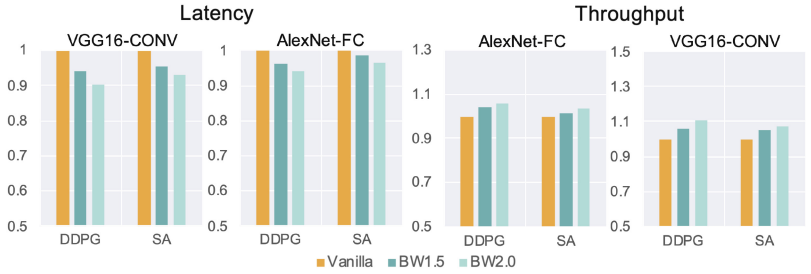

Fig. 13. Latency and throughput of the placements optimized under the original (vanilla) configuration, with changing off-chip communication bandwidth.

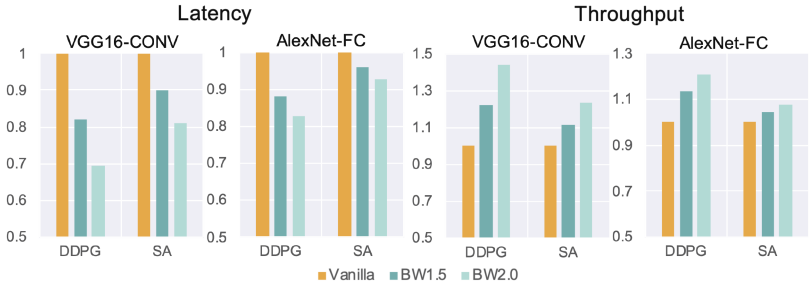

Fig. 14. Latency and throughput optimized by DDPG and SA under each different off-chip communication bandwidth.

placements found by DDPG and SA under each new configuration, to figure out their abilities of making use of different communication situations caused by different off-chip communication bandwidths. Obviously, more improvements are achieved by DDPG than those of SA: for CONV layers in VGG16, SA decreases the latency by 10% and 19%, while DDPG reduces the latency by 18% and 31%, with 1.5× and 2.0× off-chip bandwidth, respectively; for FC layers in AlexNet, SA decreases the latency by 4% and 8%, while DDPG reduces the latency by 12% and 17%, with 1.5× and 2.0× off-chip bandwidth, respectively. Generally, FC layers in AlexNet are relatively not bound by the off-chip communication bandwidth. In both of the cases here, DDPG is more capable to figure out the communication properties in the system, and find optimal placements under the corresponding configuration. Figure 12(c) shows the traffic of the placement optimized by DDPG with doubled off-chip bandwidth, where DDPG leverages the improved off-chip communication capability by subtly increasing the off-chip communication workloads and slightly alleviating the on-chip communication, compared with Figure 12(b).

We also make attempts to another case, where the number of cores per chip is decreased from 16×16 to 8×8 and so the number of chips is quadruple. In this case, resources are sacrificed for performance, i.e., adding more communication resources to release the average communication burden on each off-chip link. It is worth noting that directly reducing the number of cores per chip in the absence of modifying the previously optimized placements may cause random effects: as shown in Figure 15, some receive performance gains, while others' performance is hurt, which is mainly attributed to the possible destruction of the spatial locality in communication. After core placement optimization under the new configuration, SA attains 22% and 4% reduction in latency, while DDPG reaches 39% and 24% reduction in latency, for CONV layers in VGG16 and FC layers in AlexNet, respectively, which is illustrated in Figure 16. DDPG optimizes core placement via trials and interactions with the environment to better understand and further leverage communication

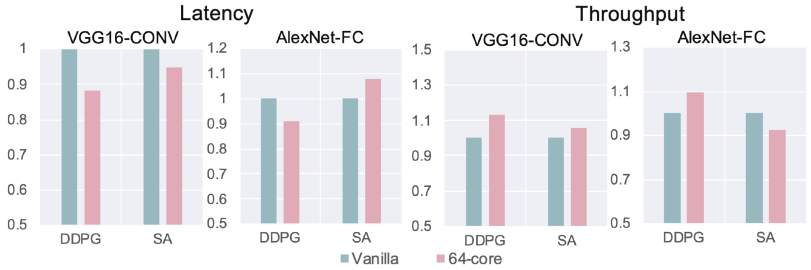

Fig. 15. Latency and throughput of the placements optimized under the original (vanilla) configuration, with changing number of cores per chip.

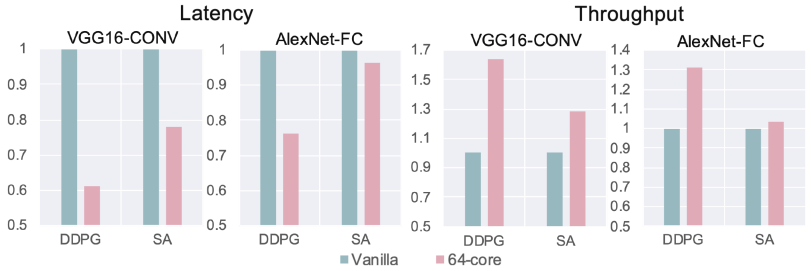

Fig. 16. Latency and throughput optimized by DDPG and SA under each different number of cores per chip.

characteristics brought from different hierarchical structures. Simultaneously, there appears better utilization of the spacial locality in communication patterns of different workloads, where logic cores obtaining more connectivity are grouped more tightly. As displayed in Figure 12(d) that exhibits the communication traffic of the placement optimized by DDPG with 8×8 cores per chip, the off-chip communication is apparently reduced and balanced, with lightweight on-chip communication; and central chips and cores are relatively busier, since packets from other cores may transit through them.

Working in a topology-agnostic manner. Besides the 2D mesh, DDPG has great versatility to deal with other topologies, such as 2D torus, HNoC [10], and dragonfly [40]. We demonstrate this by building several small multi-chip many-core systems, since these topologies may have scalability issues: for 2D torus and HNoC, there consists of a 3×3 chip array with 2×2 cores per chip; for dragonfly, there consists of six chips with five cores per chip. All other configurations are set the same as those shown in Table 1, except for the weight buffer size and the input/activation buffer size, both of which are selected as 16 KB. A synthetic MLP with 600-467-124-103 structure is taken as the workload.

Figure 17 displays the latency of different core placement methods for different topologies. Even though SA already attains good performances, it is surpassed by DDPG, where on the geometric average DDPG achieves 19%, 12%, and 8% reduction in latency, compared with BS, RS, and SA, respectively. SA is a probabilistic technique and uses meta-heuristic aiming at approximations of the global optima, which searches solutions to some extent hinging on randomness, and thus is not always reliable; whereas DDPG, which is intrinsically based on trial and error, makes a trade-off between exploration and exploitation, so it is more capable to capture the communication characteristics, i.e., domain specific information, via interactions with the environment, indicating its ability of working in a topology-agnostic manner. Additionally, through the leverage of CONV layers, DDPG is able to figure out spatial features aroused from different topologies, which is

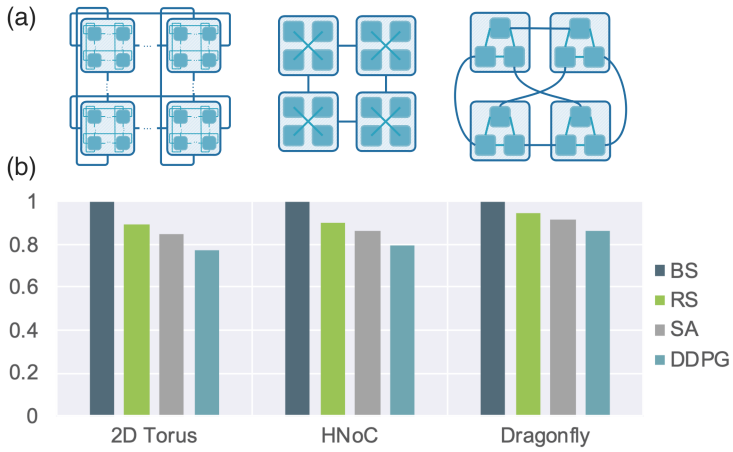

Fig. 17. Latency of different placement methods for different topologies: (a) illustration of different topologies, and (b) latency normalized to the BS.

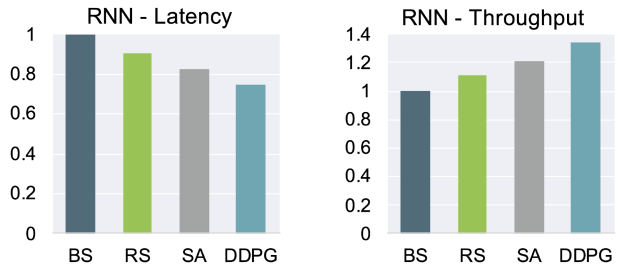

Fig. 18. Latency and throughput of different placement methods on the RNN workload, both of which are normalized to BS (the baseline).

essential and beneficial for an optimized placement; through the leverage of past experiences, DDPG has better understanding of both the system and the placement being predicted.

4.5 Discussions

Extension to RNN workloads. As current inference systems are capable to run a wide range of applications, including but not limited to vision-related tasks (most of which are CNN-based), natural language processing and recommendation systems (most of which are based on recurrent neural networks (RNNs)), we extend the evaluation of our method to RNN workloads, taking a multi-layer long short-term memory (LSTM) RNN as an example. For RNN configurations, the size of the input vector is set as 512, with two LSTM layers each of which consists of 512 neurons, and the output is a scalar generated by an FC layer; all architectural configurations are set the same as those shown in Table 1. As shown in Figure 18, DDPG can reduce the latency by 26%, 18%, 11%, and improve the throughput by 1.35×, 1.21×, 1.12×, compared with BS, RS, and SA, respectively.

Consideration of other global optimization methods. There are various global optimization methods aiming at finding the global optima in a extremely large search space. Besides the SA that is mainly considered in our work, we also give a glimpse to the genetic algorithm [87], one prominent instance of evolutionary optimization algorithms. As depicts in Figure 19, the genetic algorithm (denoted by GA) cannot beat the SA for the FC layers in AlexNet and CONV layers

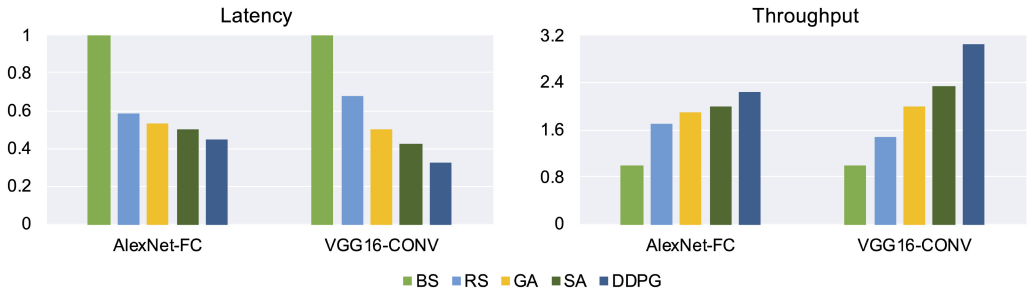

Fig. 19. Comparison with the generic algorithm.

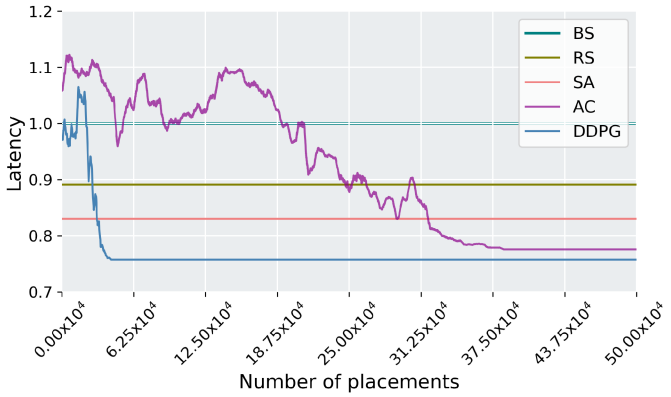

Fig. 20. Learning curves of DDPG and AC.

in VGG16. In our future work, we would like to make a thorough comparison with several other global optimization approaches.

Comparison between DPG and stochastic policy gradient. In our work, we apply the DPG (i.e., outputting the placement directly), due to its much faster convergence speed than the case using stochastic policy gradient algorithms (i.e., predicting the probability on the entire placement map). Despite the theoretical proof is aforementioned in the Section 3.2, we conduct an extra small-scale experiment as an intuitive demonstration. To make the comparison fair, we compare DDPG with its stochastic counterpart, the stochastic actor-critic (denoted by AC), and make both the actor and the critic have the same network structure as those used in the DDPG. The critic estimates the state-value function $V(s)$, and the actor generates the placement probability distribution on the entire placement map.

We consider a small multi-chip many-core system consisting of a 3×3 chip array with 2×2 cores per chip, with 16 KB as the size of weight buffer and input/activation buffer. A synthetic MLP with 435-487-227-194 is taken as the workload. Figure 20 displays learning curves of both the deterministic and the stochastic RL-based approaches, each of which is averaged on trainings with five different random number seeds and is smoothed by the moving average of neighboring 100 placements. After exploring around 50K placements, DDPG reaches its convergence, whereas AC consumes more than 375K placements to converge. This gap in convergence speed will continue to widen as the targeting multi-chip many-core system scales up, because the agent requires longer time to sufficiently explore the larger design spaces to converge.

5 RELATED WORK

We review some previous work on three major categories: the general methodology that uses ML-based methods for architecture/system designs, the target application that maps computation onto many-core systems, and the specific technique that applies deep RL to optimize latency.

ML applied for system design. Recently, there have been signs of emergence of applying ML to enhance system design, showing promising potentials. Applying ML for system design has a twofold meaning: ① the reduction of burdens on human experts designing systems manually, and ② the close of the positive feedback loop, i.e., architecture/system for ML and simultaneously ML for architecture/system, encouraging improvements on both sides. These applications include predictive performance modeling [18, 35, 44, 45, 52, 56, 90], efficient design space exploration [36, 38, 49, 92], cache replacement [5, 70, 80], prefetcher [8, 28, 93], branch prediction [25, 37], NoC design [21, 63, 85], power and resource management [4, 31], task allocation [51, 94], malware detection [15, 59], compiler design [53, 76], and so on.

Mapping computation onto many-core architecture. A series of investigations in mapping applications onto 2D mesh NoC architectures has been conducted by applying various heuristic-based techniques. They mainly target on minimizing communication energy consumption [32, 33, 68], reducing the total traffic loads and the average network hop count [69], or optimizing network latency [47, 58, 64]. There are four major differences between our work and the previous work. First, these previous approaches all focus on general-purpose many-core architectures in a single chip, while we give attention to decentralized multi-chip many-core systems for neural network workloads, where the communication related issue is caused by not only the demand for communication among different cores but also the non-uniform, hierarchical on/off-chip communication capability. It is worth noticing that in those general-purpose many-core architectures, cores are usually heterogeneous, since such architectures are designed for general workloads, whereas in our targeting multi-chip many-core architecture, cores are homogeneous and specifically designed for DNN workloads, just as shown in Figure 3. As such, our method, especially the logic cores partitioning part, is specifically tailored for the multi-chip many-core systems with DNN workloads. Second, the workloads previously considered are usually multi-media benchmarks, while we discuss neural-network-based workloads that have different communication patterns. Third, scalability is another problem of these heuristic-based methods, where they mainly handle systems with tens of cores, and it is noticed that the computation complexity grows drastically [32] as systems scale up; in contrast, our RL-based scheme is capable to deal with systems with thousands of cores. At last, previous work consider topology-specific knowledge of 2D mesh NoCs (e.g., the geometric features and communication characteristics), while our proposed method can work in a topology-agnostic manner.

Device placement optimization with RL. In recent years, there is a surge in demand on computational resources in terms of training and inference of neural networks with bigger models and larger batch sizes. One prevalent solution is to employ a heterogeneous distributed system with a mixture of different hardwares, with one instance of using the combination of CPUs and GPUs. In this scenario, the device placement refers the process of mapping the computational graph of neural networks onto hardware devices. Although such partitioning and placement decisions are usually made by human experts, there are still several concerns: first, expertise in both neural networks and hardware architectures is required; second, these decisions are often based on simple heuristics and intuitions, which do not scale well or cannot produce optimal results especially for complicated networks. To this end, Mirhoseini et al. [55] propose an RL-based method for device placement optimization, which uses a sequence-to-sequence RNN model as the parameterized policy to generate placements. This work manually groups operations and then places these

groups onto devices, and later they develop a hierarchical end-to-end model by making the manual grouping process automatic [54]. In both of their work, network parameters are trained by policy gradients via the REINFORCE [88] algorithm. In contrast, Spotlight [23] employs the proximal policy optimization (PPO) [66] to achieve better training speed and uses the softmax distributions to represent the policy. They further propose Post [22], which integrates PPO with cross-entropy minimization to acquire theoretically guaranteed optimal efficiency. Placeto [1] uses graph embeddings to encode the structure of the computational graph and exhibits good generalizability to unseen neural networks, while having high computation costs.

The work related to device placement optimization aims to achieve optimal training speed in a distributed environment with heterogeneous hardware devices. Our work focuses on optimizing inference latency in multi-chip many-core systems by core placement optimization, where we apply the deterministic actor-critic algorithm instead of pure policy gradient-based techniques and both the actor and the critic are implemented by CNNs.

6 CONCLUSION

In this work, we consider DNN inference in multi-chip many-core systems and formulate the core placement optimization problem. Previous work mainly focuses on mapping multi-media applications onto many-core architectures in a single chip and does not consider the communication-related issue in multi-chip many-core systems, which is caused by not only the demand for communication among different cores but also the non-uniform, hierarchical on/off-chip communication capability; another concern is the scalability of these heuristic-based approaches as systems scale up. To this end, we propose an RL-based method to automatically optimize core placement through DDPG, taking into account information of the environment by performing a series of trials and using CNNs to extract spatial features of different placements. We evaluate our proposed method on AlexNet, VGG16, and ResNet50, where on average DDPG reduces the overall latency by 50.5%, 38.4%, and 18.6%, and improves the throughput by 1.99×, 1.61×, and 1.22×, compared with BS, RS, and SA, respectively. We further demonstrate that our proposed RL-based method is capable of finding optimal placements, taking advantage of different communication properties caused by different system configurations, and it can also work in a topology-agnostics manner.

REFERENCES

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient progressive device placement optimization. In *Proceedings of the NIPS Machine Learning for Systems Workshop*.

[2] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. 2015. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 34, 10 (2015), 1537–1557.

[3] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731.

[4] Peter E. Bailey, David K. Lowenthal, Vignesh Ravi, Barry Rountree, Martin Schulz, and Bronis R. De Supinski. 2014. Adaptive configuration selection for power-constrained heterogeneous systems. In *Proceedings of the 43rd International Conference on Parallel Processing*. IEEE, 371–380.

[5] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing cache performance under uncertainty. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 109–120.

[6] Richard Bellman. 1957. A Markovian decision process. *J. Math. Mech.* 6, 5 (1957), 679–684.

[7] Troy Beukema, Michael Sorna, Karl Selander, Steven Zier, Brian L. Ji, Phil Murfet, James Mason, Woogeun Rhee, Herschel Ainspan, Benjamin Parker, and Michael Beakes. 2005. A 6.4-Gb/s CMOS SerDes core with feed-forward and decision-feedback equalization. *IEEE J. Solid-State Circ.* 40, 12 (2005), 2633–2645.

[8] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 1–13.

[9] Andrea Boni, Andrea Pierazzi, and Davide Vecchi. 2001. LVDS I/O interface for Gb/s-per-pin operation in 0.35-/spl mu/m CMOS. *IEEE J. Solid-State Circ.* 36, 4 (2001), 706–711.

[10] Snaider Carrillo, Jim Harkin, Liam J. McDaid, Fearghal Morgan, Sandeep Pande, Seamus Cawley, and Brian McGinley. 2012. Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations. *IEEE Trans. Parallel Distributed Syst.* 24, 12 (2012), 2451–2461.

[11] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 609–622.

[12] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE, 367–379.

[13] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.* 12 (Aug. 2011), 2493–2537.

[14] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steve McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. 2018. Loihi: A neuromorphic many-core processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99.

[15] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Comput. Architect. News* 41, 3 (2013), 559–570.

[16] Lei Deng, Ling Liang, Guanrui Wang, Liang Chang, Xing Hu, Xin Ma, Liu Liu, Jing Pei, Guoqi Li, and Yuan Xie. 2018. Semimap: A semi-folded convolution mapping for speed-overhead balance on crossbars. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 39, 1 (2018), 117–130.

[17] Lei Deng, Guanrui Wang, Guoqi Li, Shuangchen Li, Ling Liang, Maohua Zhu, Yujie Wu, Zheyu Yang, Zhe Zou, Jing Pei, Zhenzhi Wu, Xing Hu, Yufei Ding, Wei He, Yuan Xie, and Luping Shi. 2020. Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation. *IEEE J. Solid-State Circ.* 55, 8 (2020), 2228–2246.

[18] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 39–52.

[19] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

[20] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. 2016. Convolutional networks for fast energy-efficient neuromorphic computing. *Proc. Natl. Acad. Sci. U.S.A.* 113, 41 (2016), 11441–11446. DOI: https://doi.org/10.1073/pnas.1604850113

[21] Quintin Fettes, Mark Clark, Razvan Bunescu, Avinash Karanth, and Ahmed Louri. 2019. Dynamic voltage and frequency scaling in NoCs with supervised and reinforcement learning techniques. *IEEE Trans. Comput.* 68, 3 (2019).

[22] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Post: Device placement with cross-entropy minimization and proximal policy optimization. In *Advances in Neural Information Processing Systems*. MIT Press, 9971–9980.

[23] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *Proceedings of the International Conference on Machine Learning*. 1662–1670.

[24] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability*. Vol. 174. Freeman, San Francisco, CA.

[25] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. 2019. Bit-level perceptron prediction for indirect branches. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 27–38.

[26] Alex Graves and Navdeep Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 1764–1772.

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 243–254.

[28] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *Proceedings of the International Conference on Machine Learning*. 1924–1933.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[30] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.* 29, 6 (2012), 82–97.

[31] Henry Hoffmann. 2015. JouleGuard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 198–214.

[32] Jingcao Hu and Radu Marculescu. 2003. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 233–239.

[33] Jingcao Hu and Radu Marculescu. 2005. Energy-and performance-aware mapping for regular NoC architectures. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 24, 4 (2005), 551–562.

[34] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. 2019. Learning agile and dynamic motor skills for legged robots. *Sci. Robot.* 4 (2019), eaau5872.

[35] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. *Efficiently Exploring Architectural Design Spaces via Predictive Modeling*. Vol. 41. ACM.

[36] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Stargazer: Automated regression-based GPU design space exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2–13.

[37] Daniel A. Jiménez. 2011. An optimized scaled neural branch predictor. In *Proceedings of the IEEE 29th International Conference on Computer Design (ICCD'11)*. IEEE, 113–118.

[38] Ali Jooya, Nikitas Dimopoulos, and Amirali Baniasadi. 2016. MultiObjective GPU design space exploration optimization. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'16)*. IEEE, 659–666.

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, 1–12.

[40] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly scalable dragonfly topology. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 77–88.

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. MIT Press, 1097–1105.

[42] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[43] Yann LeCun, Corinna Cortes, and C. J. Burges. 2010. MNIST handwritten digit database. Retrieved from http://yann.lecun.com/exdb/mnist/.

[44] Benjamin C. Lee and David M. Brooks. 2007. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 340–351.

[45] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. 2008. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 270–281.

[46] Matthew Kay Fei Lee, Yingnan Cui, Thannirmalai Somu, Tao Luo, Jun Zhou, Wai Teng Tang, Weng-Fai Wong, and Rick Siow Mong Goh. 2019. A system-level simulator for RRAM-based neuromorphic computing chips. *ACM Trans. Arch. Code Optim.* 15, 4 (2019), 1–24.

[47] Tang Lei and Shashi Kumar. 2003. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Proceedings of the Euromicro Symposium on Digital System Design*. IEEE, 180–187.

[48] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. ArXiv:1509.02971. Retrieved from https://arxiv.org/abs/1509.02971.

[49] Ting-Ru Lin, Yunfan Li, Massoud Pedram, and Lizhong Chen. 2019. Design space exploration of memory controller placement in throughput processors with deep learning. *IEEE Comput. Arch. Lett.* 18, 1 (2019), 51–54.

[50] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3431–3440.

[51] Shiting Justin Lu, Russell Tessier, and Wayne Burleson. 2015. Reinforcement learning for thermal-aware many-core task allocation. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*. ACM, 379–384.

[52] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Proceedings of the International Conference on Machine Learning*. 4505–4515.

[53] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. Compiler auto-vectorization with imitation learning. In *Advances in Neural Information Processing Systems*. MIT Press, 14598–14609.

[54] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *Proceedings of the 35th International Conference on Machine Learning*. JMLR. org.

[55] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2430–2439.

[56] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. 2017. Esp: A machine learning approach to predicting application interference. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC'17)*. IEEE, 125–134.

[57] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. ArXiv:1312.5602. Retrieved from https://arxiv.org/abs/1312.5602.

[58] Srinivasan Murali and Giovanni De Micheli. 2004. Bandwidth-constrained mapping of cores onto NoC architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2. IEEE, 896–901.

[59] Meltem Ozsoy, Khaled N. Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2016. Hardware-based malware detection using low-level architectural features. *IEEE Trans. Comput.* 65, 11 (2016), 3332–3344.

[60] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R. Lester, Andrew D. Brown, and Steve B. Furber. 2013. SpiNNaker: A 1-W 18-core system-on-chip for massively parallel neural network simulation. *IEEE J. Solid-State Circ.* 48, 8 (2013), 1943–1953.

[61] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, 27–40.

[62] Jing Pei, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, Feng Chen, Ning Deng, Si Wu, Yu Wang, Yujie Wu, Zheyu Yang, Cheng Ma, Guoqi Li, Wentao Han, Huanglong Li, Huaqiang Wu, Rong Zhao, Yuan Xie, and Luping Shi. 2019. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* 572, 7767 (2019), 106–111.

[63] Nishant Rao, Akshay Ramachandran, and Amish Shah. 2018. MLNoC: A machine learning based approach to NoC design. In *Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'18)*. IEEE, 1–8.

[64] Pradip Kumar Sahu, Nisarg Shah, Kanchan Manna, and Santanu Chattopadhyay. 2010. A new application mapping algorithm for mesh based network-on-chip design. In *Proceedings of the Annual IEEE India Conference (INDICON'10)*. IEEE, 1–4.

[65] Ahmad E. L. Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. 2017. Deep reinforcement learning framework for autonomous driving. *Electronic Imag.* 2017, 19 (2017), 70–76.

[66] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. ArXiv:1707.06347. Retrieved from https://arxiv.org/abs/1707.06347.

[67] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 14–27.

[68] Pradeep Kumar Sharma, Santosh Biswas, and Pinaki Mitra. 2019. Energy efficient heuristic application mapping for 2D mesh-based network-on-chip. *Microprocess. Microsyst.* 64 (2019), 88–100.

[69] Wein-Tsung Shen, Chih-Hao Chao, Yu-Kuang Lien, and An-Yeu Wu. 2007. A new binomial mapping and optimization algorithm for reduced-complexity mesh-based on-chip network. In *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS'07)*. IEEE, 317–322.

[70] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 413–425.

[71] Dongjoo Shin, Jinmook Lee, Jinsu Lee, and Hoi-Jun Yoo. 2017. 14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'17)*. IEEE, 240–241.

[72] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML'14)*, Volume 32. I-387–I-395.

[73] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.

[74] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. ArXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556.

[75] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards hybrid parallelism for deep-learning accelerator array. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, 56–68.

[76] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using machine learning to improve automatic vectorization. *ACM Trans. Arch. Code Optim.* 8, 4 (2012), 50.

[77] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*. MIT Press, 3104–3112.

[78] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

[79] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*. MIT Press, 1057–1063.

[80] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. 2016. Perceptron learning for reuse prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[81] George E. Uhlenbeck and Leonard S. Ornstein. 1930. On the theory of the Brownian motion. *Phys. Rev.* 36, 5 (1930), 823.

[82] Gianvito Urgese, Francesco Barchi, Enrico Macii, and Andrea Acquaviva. 2016. Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms. *IEEE Trans. Emerg. Topics Comput.* 6, 3 (2016), 317–329.

[83] Peter J. M. Van Laarhoven and Emile H. L. Aarts. 1987. Simulated annealing. In *Simulated Annealing: Theory and Applications*. Springer, 7–15.

[84] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019), 350–354.

[85] Ke Wang, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. 2019. IntelliNoC: A holistic design framework for energy-efficient and reliable on-chip communication for many cores. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 589–600.

[86] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Mach. Learn.* 8, 3–4 (1992), 279–292.

[87] Darrell Whitley. 1994. A genetic algorithm tutorial. *Stat. Comput.* 4, 2 (1994), 65–85.

[88] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* 8, 3–4 (1992), 229–256.

[89] John M. Wilson, Walker J. Turner, John W. Poulton, Brian Zimmer, Xi Chen, Sudhir S. Kudva, Sanquan Song, Stephen G. Tell, Nikola Nedovic, Wenxu Zhao, Sunil R. Sudhakaran, C. Thomas Gray, and William J. Dally. 2018. A 1.17 pJ/b 25Gb/s/pin ground-referenced single-ended serial link for off-and on-package communication in 16nm CMOS using a process-and temperature-adaptive voltage regulator. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'18)*. IEEE, 276–278.

[90] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 564–576.

[91] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. 2018. DNN dataflow choice is overrated. *ArXiv preprint arXiv:1809.04070*. Retrieved from https://arxiv.org/abs/1809.04070.

[92] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. 2013. Towards machine learning-based auto-tuning of mapreduce. In *Proceedings of the IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 11–20.

[93] Yuan Zeng and Xiaochen Guo. 2017. Long short-term memory–based hardware prefetcher: A case study. In *Proceedings of the International Symposium on Memory Systems*. ACM, 305–311.

[94] Haitao Zhang, Bingchang Tang, Xin Geng, and Huadong Ma. 2018. Learning driven parallelization for large-scale video workload in hybrid CPU-GPU cluster. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 32.

[95] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 20.

[96] Brian Zimmer, Rangharajan Venkatesan, Yakun Sophia Shao, Jason Clemons, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Stephen W. Keckler, and Brucek Khailany. 2019. A 0.11 pJ/Op, 0.32-128 TOPS, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16nm. In *Proceedings of the Symposium on VLSI Circuits*. IEEE, C300–C301.

Received April 2020; revised July 2020; accepted August 2020